Visual Analytics for Understanding the Evolution of Large Software Projects

A. Telea[1] and L. Voinea[2]

[1] Institute for Math. and Computer Science, University of Groningen, the Netherlands
[2] SolidSource BV, Eindhoven, the Netherlands

Abstract. We present how a combination of static source code analysis, repository analysis, and visualization techniques has been used to effectively get and communicate insight in the development and project management problems of a large industrial code base. This study is an example of how visual analytics can be effectively applied to answer maintenance questions in the software industry.

1 Introduction

Industrial software projects encounter bottlenecks due to many factors: improper architectures, exploding code size, bad coding style, or suboptimal team structure. Understanding the causes of such problems helps taking corrective measures for ongoing projects or choosing better development and management strategies for new projects.

We present here a combination of static data analysis and visualization tools used to explore the causes of development and maintenance problems in an industrial project. The uncovered facts helped developers to understand the causes of past problems, validate earlier suspicions, and assisted them in deciding further development. Our solution fits into the emerging *visual analytics* discipline, as it uses information visualization to support the analytical reasoning about data mined from the project evolution.

2 Software Project Description

The studied software was developed in the embedded industry by three teams located in Western Europe, Eastern Europe, and India. All code is written in Keil C166, a special C dialect with constructs that closely supports hardware-related operations [2]. The development took six years (2002-2008) and yielded 3.5 MLOC of source code (1881 files) and 1 MLOC of headers (2454 files) in 15 releases. In the last 2 years, it was noticed that the project could not be completed on schedule, within budget, and that new features were hard to introduce. The team leaders were not sure what went wrong, so an investigation was performed at the end.

3 Analysis Method

The investigation had two parts: a process analysis and a product (code) analysis. We describe here only the latter. We analyzed the source code, stored in a Source Control

Management (SCM) system, using a C/C++ analyzer able to handle incorrect and incomplete code by using a flexible GLR grammar [3], and a tool to mine data from SCM systems. We were able to easily modify this grammar (within two work days) to make it accept the Kyle C dialect. The analyzer creates a fact database that contains static constructs *e.g.* functions and dependencies, and several software metrics.

We applied the above process to all files in all 15 releases. Given the high speed of the analyzer, this took under 10 minutes on a standard Linux PC. After data extraction, we used several visualizations to make sense of the extracted facts. Each visualization looks at different aspects of the code, so their combination aims to correlate these aspects in one coherent picture. The visualizations (presented next) were interactively shown to the developers, who were asked to comment on the findings, their relevance, and their usefulness for understanding the actual causes of the development problems.

3.1 Modification Request Analysis

Figure 1 shows two images of the project structure, depicted as a three-level treemap (packages, modules and files). Treemaps have been used in many software visualization applications to show metrics over structure [1]. The smallest rectangles are files, colored by various metrics, and scaled by file size (LOC)[3]. The left image shows the number of modification requests (MRs) per file. Files with more than 30 MRs (red) appear spread over all packages. The right image shows the same structure, colored by team identity. We see that most high-MR files (red in the left image) are developed by Team A (red in the right image), which is located in India. As this team had communication issues with the other two teams, a better work division may be to reassign Team A to work on a single, low-MR-rate, package.

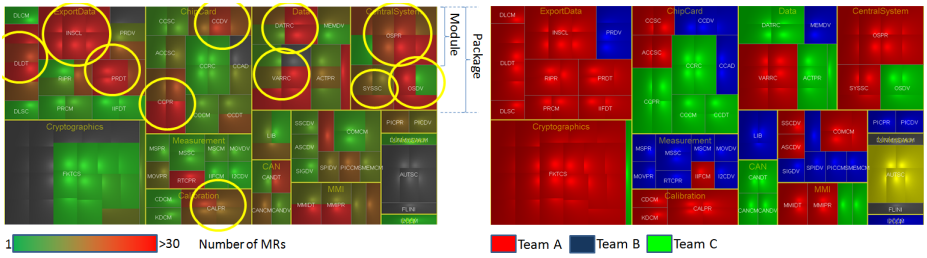

Fig. 1. Team assessment: number of MRs (left) and team structure (right)

We further analyzed the MR records. Figure 2 left shows the MR distribution over project and time. Files are drawn as gray horizontal pixel lines, stacked in creation order from bottom to top. The x axis shows time (years 2002 to 2008). Red dots show the location (file, time) of the MRs. We see that less than 15% of the files have been created in the second project half, but most MRs in this half address *older* files, so the late work mainly tried to fix old requirements. The right image supports this hypothesis: each horizontal bar indicates the number of file changes related to one MR range (MRs are functionally grouped per range), the x axis shows again time. We see that older MRs (at top) have large activity spreads over time. For example, in mid-2008, developers still try to address MRs introduced in 2005-2006. Clearly, new features are hard to introduce if most work has to deal with old MRs.

[3] We recommend viewing this paper in full color

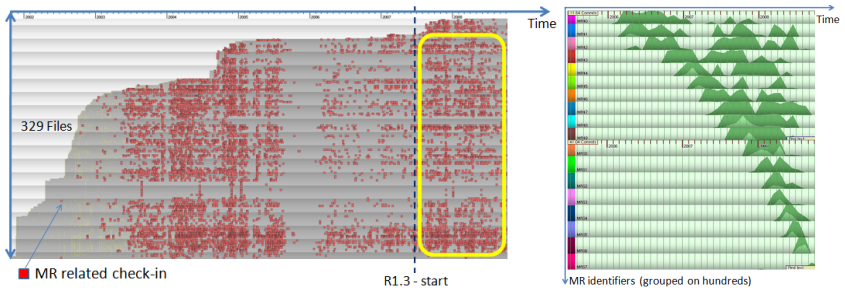

Fig. 2. MR evolution per file (left) and per range of MRs (right)

Figure 3 shows the MRs versus project structure. The left image shows MR criticality (green=low, red=high). We see a correlation with the MR count and team distribution (Fig. 1): many critical MRs are assigned to Team A, which had communication problems. The right image indicates the average closure time of a MR: critical MRs, involving Team A, took long to close. This partially explains the encountered delays and further supports the idea of reassigning critical MRs to other teams.

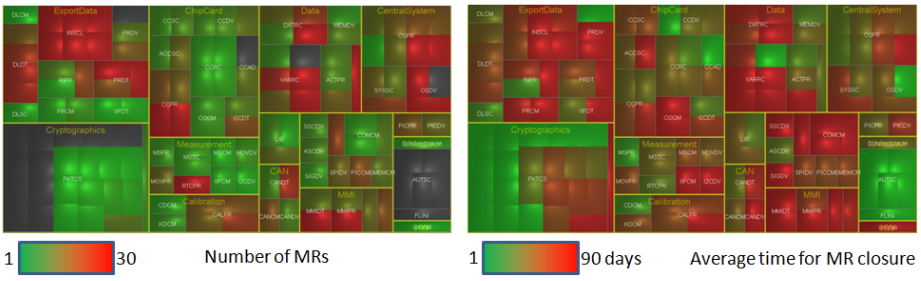

Fig. 3. MR criticality: MRs vs project structure (left); MR closure time (right)

3.2 Structural Analysis

We next analyzed the evolution in time of several software metrics (see Fig. 4). The graphs show a relatively low increase of project size (functions and function calls) and, roughly, and overall stable dependency count (fan-in) and code complexity. The fan-out and number of function calls increases more visibly. Hence, we do not think that maintenance problems were caused by an explosion of code size or complexity, as is the case in other projects. Additional analyses such as call and dependency graphs (not shown here) confirmed, indeed, that the system architecture was already well-defined and finalized in the first project half, and that the second half did barely change it.

4 Conclusions

Correlating the MR analysis with the static code analysis, our overall conclusion is that the most work in this project was spent in finalizing early requirements. Correlating with the developer interviews, this seems to be mainly caused by a suboptimal allocation of teams to requirements and packages. The project owners stated that they found this type

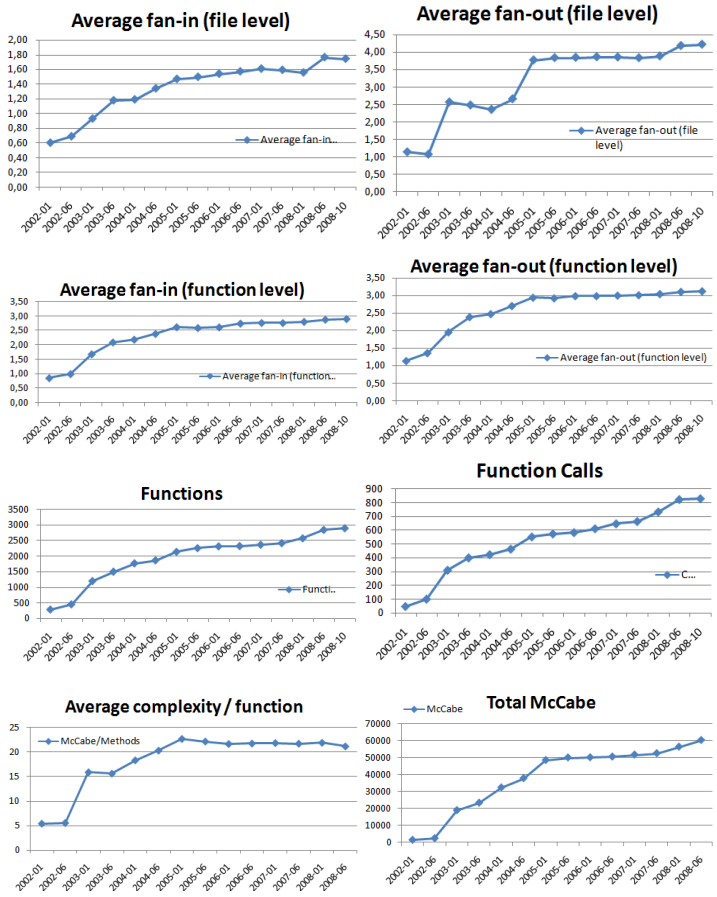

Fig. 4. Evolution of static analysis metrics

of analysis very useful from several perspectives: supporting earlier suspicions, providing concrete measures for assessing the project evolution, presenting these measures in a simple and easy way for discussion, and supporting further project management decisions. An attractive element was the short duration of the analysis: the entire process lasted three days, including the developer interviews, code analysis, and results discussion. Also, the stakeholders stated that using such types of analyses continuously, and from the beginning of, new projects is of added value in early detection and discussion of problems.

References

1. M. Balzer and O. Deussen. Voronoi treemaps for the visualization of software metrics. In *Proc. ACM Softvis*, pages 165–172, 2005.
2. Keil, Inc. The Keil C166 compiler. 2008. `http://www.keil.com`.
3. A. Telea and L. Voinea. An interactive reverse engineering environment for large-scale C++ code. In *Proc. ACM SoftVis*, pages 67–76, 2008.